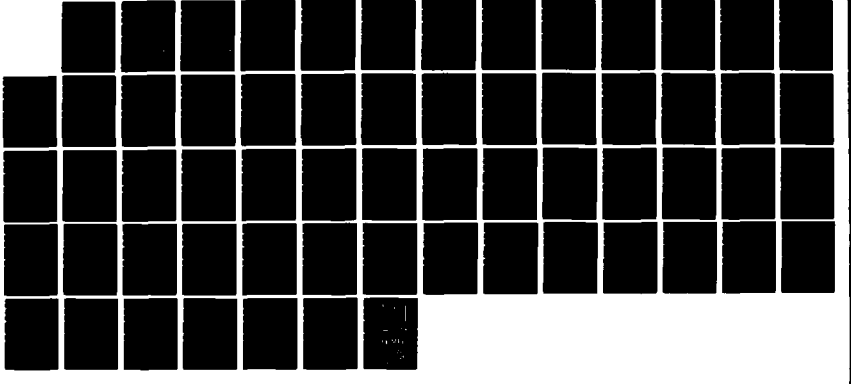
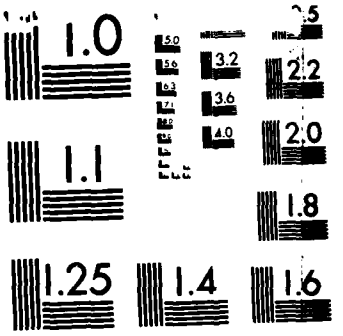
AD-A163 505 IMS DESIGN--ALTERNATIVES ANALYSIS AND STRATEGIES REVISION(U) ALFRED P SLOAN SCHOOL OF MANAGEMENT CAMBRIDGE MA CENTER FOR I... M J ABRAHAM DEC 85 TR-17 1/1

UNCLASSIFIED N00039-80-K-0498 F/G 9/2 NL





IMS Design--
Alternatives, Analysis, and Strategies

Michael J. Abraham

Technical Report #17

Draft--April 1982
Modified--December 1985

Center for Information Systems Research
Massachusetts Institute of Technology
Sloan School of Management
77 Massachusetts Avenue
Cambridge, Massachusetts, 02139

Contract Number N00039-80-K0498
Internal Report Number M010-8512-17

IMS Design--
Alternatives, Analysis, and Strategies

Michael J. Abraham

Technical Report #17

Draft--April 1982
Modified--December 1985

Principal Investigator:
Professor S. E. Madnick

Prepared for:
Naval Electronics Systems Command
Washington, D.C.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER Technical Report #17	2. GOVT ACCESSION NO. AD-A163505	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) IMS Design--Alternatives, Analysis, and Strategies		5. TYPE OF REPORT & PERIOD COVERED
		6. PERFORMING ORG. REPORT NUMBER M010-8512-17
7. AUTHOR(s) Michael J. Abraham		8. CONTRACT OR GRANT NUMBER(s) N00039-80-K0498
9. PERFORMING ORGANIZATION NAME AND ADDRESS Center for Information Systems Research Sloan School of Management, M.I.T. Cambridge, MA 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Electronics Systems Command		12. REPORT DATE December 1985
		13. NUMBER OF PAGES 56
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Data storage hierarchy, data base management system, hierarchical decomposition, distributed control, data location

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Modern organizations are becoming increasingly reliant on the storage and processing of very large data bases in support of their accounting, operational control, and high-level decision-making functions. It is expected that future high-performance Data Base Management Systems will be required to provide storage capacities and transaction rates several orders of magnitude greater than those of any current systems. As

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

DBMS's become ever more integral parts of many organizations' operations the costs of system failure or unavailability increase correspondingly. Accordingly, there is a growing need for "fault tolerant" systems which can provide continuous availability in the presence of many types of internal component failure (both hardware and software).

To meet the requirements for increased speed, capacity, and availability the IMS Data Base Computer (INFOPLEX) employs a highly parallel, distributed control architecture. The preliminary INFOPLEX design consists of two logically and physically separate components: a physical storage hierarchy and a functional hierarchy. The general structure of an INFOPLEX data storage hierarchy has been developed and a preliminary set of control algorithms has been proposed.

This report presents a refind architectural specification and outlines a set of control algorithms which take advantage of theoretical properties of INFOPLEX-like storage systems. Section 2 discusses the overall rationale for the INFOPLEX hierarchical design concept. Section 3 presents the details of the design issues and tradeoffs related to performance and reliability, and developes a general strategy for efficient READ/WRITE control and reliable fault handling. Finally, Section 4 summarizes this report, and indicates the directions for further research.

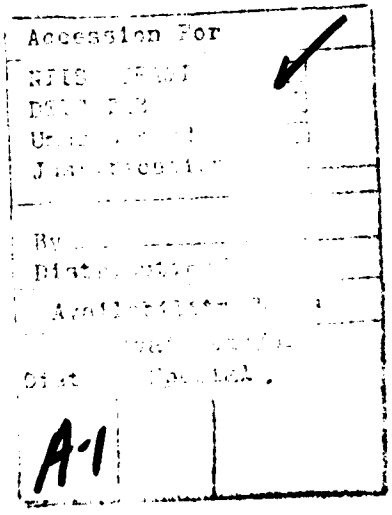


Table of Contents

1 INTRODUCTION

Modern organizations are becoming increasingly reliant on the storage and processing of very large data bases in support of their accounting, operational control, and high-level decision-making functions. Contemporary Data Base Management Systems (DBMS's) are capable of handling data bases on the order of a trillion (10^{12}) bits of data [Simo75], and can process transactions at rates of up to 100 queries per second [Abe77]. However, it is expected that future high-performance DBMS's will be required to provide storage capacities and transaction rates several orders of magnitude greater than those of any current systems. It is not unreasonable to project requirements of a quadrillion (10^{15}) bits and one million database accesses per second [Madn77].

As DBMS's become ever more integral parts of many organizations' operations the costs of system failure or unavailability increase correspondingly. Accordingly, there is a growing need for "fault tolerant" systems which can provide continuous availability in the presence of many types of internal component failure (both hardware and software).

To meet the requirements for increased speed, capacity, and availability the IMS Data Base Computer (INFOPLEX) employs a highly parallel, distributed control architecture. The preliminary INFOPLEX design consists of two logically and physically separate components: a physical storage hierarchy which consists of a series of micro-processor controlled storage devices functioning as a very large (10^{15} bits) virtual memory; and a functional hierarchy, consisting of a series of micro-processor clusters, which provide user interfaces, security, and memory management facilities for an INFOPLEX database system [Hsu80]. The general structure of an INFOPLEX data storage hierarchy has been developed and a preliminary set of

control algorithms has been proposed [Lam79a].

This report presents a refined architectural specification and outlines a set of control algorithms which take advantage of theoretical properties of INFOPLEX-like storage systems [Lam79b, Abra79]. Section 2 discusses the overall rationale for the INFOPLEX hierarchical design concept. Section 3 presents the details of the design issues and tradeoffs related to performance and reliability, and develops a general strategy for efficient READ/WRITE control and reliable fault handling. Finally, Section 4 summarizes this report, and indicates the directions for further research.

2 OVERVIEW OF DSH-III DESIGN

2.1 Introduction

The INFOPLEX Data Storage Hierarchy III (DSH-III) is a model for a very large, high-speed, reliable storage system. The primary design objective of DSH-III is to provide a user (in particular the INFOPLEX Functional Hierarchy) with a very large, high-speed virtual address space. As will be seen, DSH-III takes complete responsibility for all physical data management, control of the various storage devices in the system, and recovery from almost all types of single-component failures. By incorporating the intelligence needed to perform these functions into the storage system, DSH-III is able to provide a user with a very simple, clean, easy to use interface. In particular, DSH-III can provide almost complete memory system functionality to a user through two primitive operations -- READ and WRITE.

While this paper presents the details of only two primitive operations -- READ and WRITE -- additional primitives can be added as experience indicates in order to increase the usability and flexibility of DSH-III. A partial list of such primitives might include:

TEST_AND_SET - provides an atomic conditional update operation which can be used to support P and V [Dijk68] synchronization operations;

SET_SECURE - allows a user to select a portion of the DSH-III virtual address space for special high-reliability handling, such as automatic replication and duplication of data. This facility might be used to protect data of an especially critical nature,

such as system control tables. The Tandem [Bart77] computer is an example of a system which uses automatic duplication to enhance availability;

BLOCK_MOVE - allows a user to transfer large data blocks from one location in virtual memory to another. By simply modifying the mapping of real to virtual memory, this operation could be accomplished without any actual data movement;

BLOCK_ZERO - allows a user to initialize (set to zeroes) a large area of virtual memory.

2.2 Basis of DSH-III Design

The fundamental rationale for the design of DSH-III to be presented in this paper is based on three principles: 1) employing a range of storage technologies, 2) taking advantage of locality of reference, and 3) hierarchical decomposition and distributed control.

2.2.1 Range of Storage Technologies

A basic problem which constrains the design of any high-speed, high-capacity storage system is that no single storage technology can meet the requirements for both speed and capacity within reasonable cost constraints. For example, high-speed semi-conductor RAM can support random access times of 50ns, or less, but costs on the order of 5 cents per byte. At the other end of the cost/capacity/speed spectrum are mass storage devices such as automated tape handlers, which can store large quantities of data at a cost of only .0005 cents per byte, but which have access times up to seven orders of magnitude slower than high-speed RAM. In between these

two extremes are a range of storage device technologies as shown in Table 2.1.

Storage Level	Example	Random Access Time	Sequential Transfer Rate (bytes/sec)	Unit Capacity (bytes)	Unit Price (cents/byte)
1. Cache	HMOS RAM	40-50 ns	80M (4 bytes/access)	32K	5
2. Main	NMOS RAM	500 ns	8M (4 bytes/access)	1M	0.1
3. Backing	High-Speed Drums	2 ms	2M	10M	0.5
4. Secondary	Disks	25 ms	1M	1-1.5B	0.005
5. Mass	Automated Tape Handlers	1 sec	1M	100B	0.0005

Table 2.1 - Summary of Storage Technologies

The approach taken in the design of the INFOPLEX Data Storage Hierarchy is to utilize a range of storage technologies, with the bulk of the data stored on inexpensive but relatively slow devices and automatically migrated to higher speed devices when it is accessed. This approach is logically equivalent to that used by cache based computer systems such as the IBM 3033 [IBM3033] and by mass storage systems such as the IBM 3850 [IBM3850].

2.2.2 Locality of Reference

In order for a multimedia storage system to take full advantage of its higher speed storage devices, it is desirable that the higher speed devices be accessed relatively more often than the lower speed devices. For this goal to be attainable, it is necessary that the database be subject to a

non-homogeneous reference pattern. This non-homogeneity can be spatial or temporal and any database system which has this property is said to exhibit locality of reference [Madn73]. Spatial locality refers to access patterns for which reference to any particular data item increases the probability that related data items will also be accessed. There are many examples of this phenomenon:

- sequential flow of control in a software module implies that reference to an instruction in program storage presages references to following instructions.
- reference to a particular field in a record stored in a file system is usually accompanied by references to other fields in the same record (or the same field in related records).

Temporal locality refers to access patterns for which consecutive references to data items are correlated in time. Automatic teller systems provide a typical example of this phenomenon. A common usage consists of a balance inquiry followed by a cash withdrawal, resulting in two accesses to the account balance within a short period of time. (In fact, the withdrawal alone, or any other database update, exhibits temporal locality due to the need to read the data item to be updated before writing the modified version of the data.)

INFOPLEX takes advantage of the fact that database systems do exhibit locality [Rodr76, Robi79], and that locality can be used to increase the relative utilization of the higher speed storage devices in the system.

There are three strategies for taking advantage of locality:

- static, where high-usage data, such as key system tables, is permanently allocated to higher speed devices
- manual, where it is the responsibility of the programmer to move data to higher speed devices when it is needed
- and automatic, where the system automatically migrates high usage data to fast devices and low usage data to slow devices.

One design strategy of INFOPLEX is to use automatic migration to take

advantage of locality. This technique has the following advantages:

- it allows dynamic response to changing application loads and time-varying database content
- it relieves the programmer or system designer of the burden of allocating the data.

This strategy is analogous to that used by the Multics operating system which automatically migrates pages of virtual memory between high speed paging devices (e.g., drums) and lower speed devices (e.g., disks) in response to changes in the working sets of the active tasks in the system [Gree75]. The precise manner in which automatic migration is implemented in DSH-III is described in Section 3.

2.2.3 Hierarchical Decomposition and Distributed Control

INFOPLEX organizes its heterogeneous array of storage devices using the principle of hierarchical decomposition and distributed control. The use of hierarchical decomposition leads to a conceptual system design such as that shown in Figure 2.1. This structure has been shown [Madn79] to represent an efficient and effective method for integrating heterogeneous storage devices into a single system. There are three primary advantages to this structure.

First, the hierarchical structure supports the types of direct interlevel data transfer which are used by read, write, and automatic migration algorithms. This avoids the overhead associated with the indirect data path (i.e., drum to main memory to disk) used by the page migration scheme of Multics mentioned above.

Support for direct inter-level data transfer is a reflection of the second advantage of the hierarchical structure, namely that it facilitates the utilization of a distributed control strategy for the system. By this we mean that the basic system control and interlevel communication functions

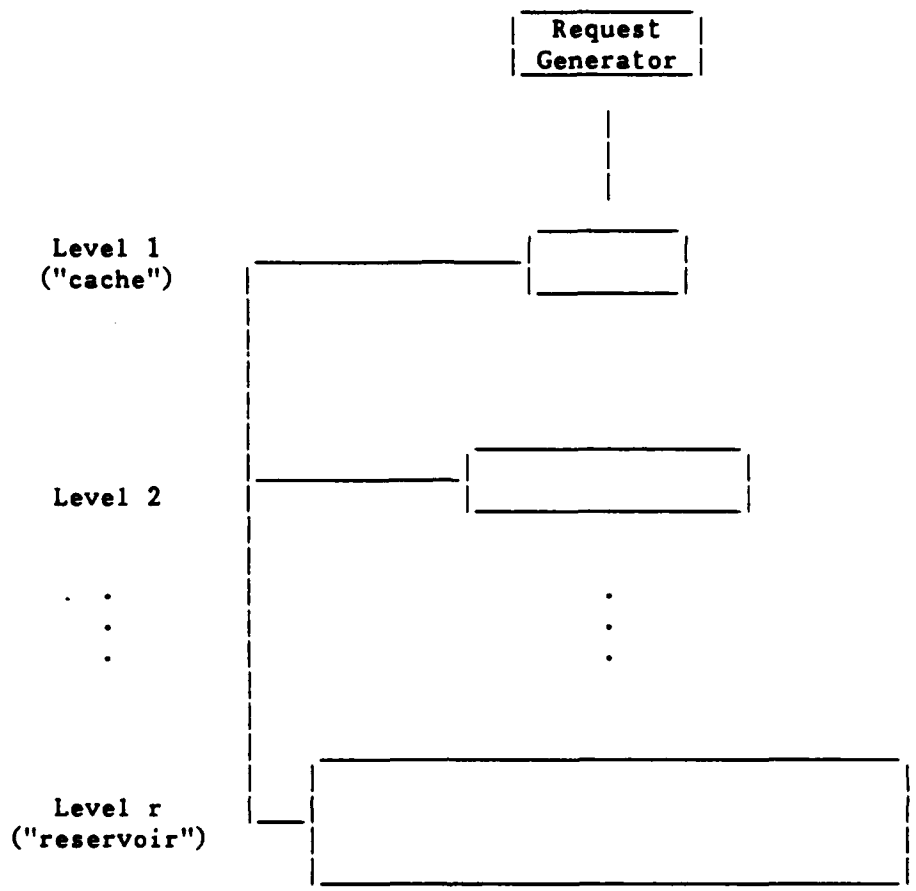


Figure 2.1 - Logical Structure of a Storage Hierarchy

will be performed by micro-processor clusters within each level. This strategy improves system performance by facilitating parallel and asynchronous operation within the hierarchy, as well as eliminating the potential reliability exposure that would be associated with a single controlling processor cluster.

Third, the design is inherently modular. This has four principal advantages

- the structure allows the use of common algorithms and software modules at each of the levels. This facilitates software design, especially in the area of interlevel communications protocols.

- if a level fails, the remaining levels form a system which is logically equivalent to the original (unfailed) system. This has important implications for the ability of the system to continue operation in the presence of failures.

- the modular structure facilitates the incorporation of new storage technologies into an INFOPLEX system. Thus the basic design should be relatively insensitive to the rapidly changing technology in this area.

- the structure allows the building of storage hierarchies with the number of levels and the types of storage device at each level customized to a particular application.

3 DESIGN OF AN INFOPLEX STORAGE HIERARCHY

This section presents the cost/performance/complexity/reliability tradeoffs and other issues underlying the design of DSH-III. We begin, in Section 3.1, with a general overview of the system topology implied by the discussion in Section 2.2. Next, Section 3.2 describes the interface between DSH-III and a user. Section 3.3 presents a justification for the data management strategies used in DSH-III. Based on these strategies, a specification for the READ algorithms used by DSH-III is developed in Section 3.4. The basic design of DSH-III is completed by the specification of WRITE algorithms for DSH-III in Section 3.5.

3.1 Overview of System Topology

Following the reasoning presented in the preceding sections leads to a conceptual system design such as the one shown in Figure 3.1. The system consists of r storage levels, Level 1 to Level r, with Level 1 containing high-speed cache memory and Level r (the "reservoir") containing mass storage devices which contain a copy of all the data in the database. One of the modules shown in Levels 2 to r is a Local Storage System (LSS), which consists of the physical devices holding the DSH-III database. LSS technology will vary from level to level, with higher levels using faster, but lower capacity, devices. In particular, because of the high response time requirements for the highest level, Level 1 will use the same storage technology for both Local Memory (LM) and LSS. For this reason, no separate LSS is shown for Level 1 in Figure 3.1.

All interlevel communication is performed via the Global Bus (GBUS). The major disadvantage of using a single GBUS instead of an inter-level bus between each adjacent pair of levels is that the parallelism of the system

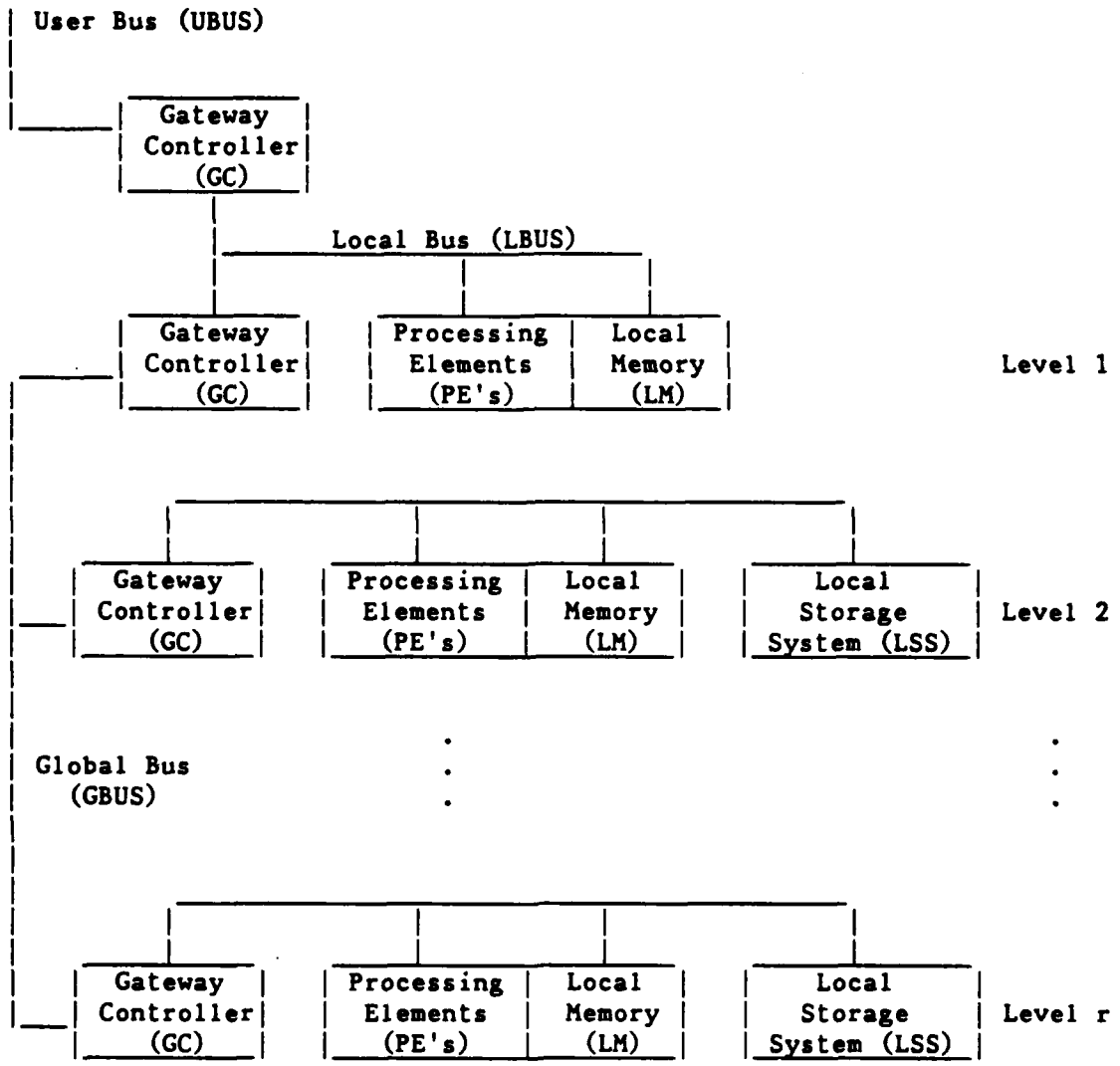


Figure 3.1 - Conceptual Structure of a Storage Hierarchy

is reduced and thus bus contention and resultant queuing delays are increased. That this is an important consideration is shown by simulation studies of a storage hierarchy [Lam79a] which showed GBUS utilizations of over 80%. While 80% utilization of the GBUS has little impact on system performance (because the GBUS has a high bandwidth relative to other components of the system) it is clear that the GBUS could easily become a bottleneck under slightly different assumptions for component speeds or

transaction loads. On the other hand, the logical GBUS could be implemented as multiple physical buses. In this case, adding extra capacity to a bottlenecked GBUS would not involve any great difficulty, but could be accomplished by replicating existing hardware structures. The potential disadvantage of using a single logical GBUS is clearly outweighed by the advantages offered by this structure. These advantages include

- the ability for a level to communicate with any other level, thus supporting "broadcasting" of information. As will be seen in Section 3.3, broadcasting greatly improves the efficiency of data movement algorithms and eases system control problems by providing a means of synchronizing operations at different levels.

- increased availability, since the system structure facilitates "graceful degradation" and continued operation in the presence of level failures.

- more cost effective utilization of resources since the resource (the GBUS) is shared by all users (the levels) in the system. This facilitates the matching of bus capacity and demand.

Each storage level consists of a number of storage devices and processing modules, interconnected via a Local Bus (LBUS). The interface between each level and the GBUS is provided by a Gateway Controller (GC) at each level. From the viewpoint of a GC, each level appears as an identical black box, i.e., the number and types of processors and storage devices within each level are transparent to the GC at that level. This conforms with the concept of modularity and commonality of algorithms and software discussed in Section 2.2.3.

A Pended Bus Protocol [Toon80] will be used for all buses in order to support the large number of devices on each bus.

A forthcoming report will present a more complete discussion of a possible hardware implementation of this hierarchical structure.

3.2 User Interface

Level 1 (the "cache" level) of the storage hierarchy serves as the interface between the user (the lowest level of the INFOPLEX Functional Hierarchy) and DSH-III. In particular this implies that Level 1 represents a shared cache structure, rather than a per user processor cache structure. Detailed simulation studies show that this single-bus, shared cache structure is a very effective and efficient topology for providing high-speed, parallel, multi-processor access to the storage hierarchy. This structure, which is made feasible by the use of the Pended Bus protocol, has the significant advantage of greatly simplifying the cache consistency control problem. For a complete discussion of possible alternate topologies and the tradeoffs among performance and consistency control policies the reader is referred to [Abde81].

In addition to the usual GC at Level 1, there is another Gateway Controller which serves as an interface between the Level 1 LBUS and a User Bus (UBUS) which connects DSH-III and the Functional Hierarchy.

From the point of view of a user, DSH-III appears as a very large linear address space which is accessed via simple primitives such as

READ(request_id,virtual_address) and

WRITE(request_id,virtual_address,data).

As noted previously, the interface has been kept as simple as possible, and a user is completely isolated from the details of data management and error recovery. The intelligence to perform these functions is distributed throughout DSH-III.

The READ and WRITE operations are not atomic operations. This means that control is returned to a user after he issues one of these commands, but before the operation has completed. When the operation finally

completes, the user is notified. This implies that a user may have multiple operations active simultaneously. Because of this, it is important to specify now DSH-III will behave if READs and WRITEs are overlapped. For sequences such as

READ, WRITE, READ complete, WRITE complete, or

WRITE, READ, WRITE complete, READ complete

DSH-III will arbitrarily treat the entire READ operation as having either preceded the WRITE or followed the WRITE completion. Therefore, the results of the READ may or may not reflect the results of the WRITE. The only guaranteed way to ensure that a READ will reflect the results of a WRITE is to issue the READ after the WRITE has completed.

The remainder of this chapter is devoted to explaining and justifying the data movement strategies used by DSH-III. The details of the algorithms used by DSH-III to support the READ and WRITE primitives, will be presented in a future report.

3.3 Data Movement Strategies

The objectives of the data movement strategies used by DSH-III are three-fold. First, the strategies attempt to take advantage of locality by migrating high usage data to the higher speed storage devices. Second, the strategies attempt to minimize unnecessary data movement within the system in order to reduce bus contention. Third, redundant copies of data blocks are maintained at various levels. This has the effect of increasing system reliability while greatly simplifying page migration algorithms.

The basic design decisions underlying the data movement strategies of DSH-III are

- when should a block of data be moved from a lower level to a higher level of the hierarchy?

- if the transfer of a data block to a higher level necessitates the removal of a block already in the higher level, how should the block to be removed be selected, and what should be done with it (e.g., should it be discarded or transferred to some other level)?

- how big should the basic unit of access and transfer (the "page size") be; should it be the same at all levels or differ from level to level?

3.3.1 Demand Paging with Replacement

The data retrieval algorithms of DSH-III are based on a demand fetch policy. Under this policy, a data block is moved from a lower to a higher level in the hierarchy only in response to an explicit READ request by a user. In other words, there is no attempt made to anticipate future retrieval requests (based on known usage patterns or locality considerations) by pre-fetching data blocks before they are explicitly requested. This does not mean that the system does not take advantage of spatial locality. As will be seen, DSH-III blocks data into pages of various sizes, and this policy does result in anticipatory retrieval of data stored in the same page as the data being explicitly retrieved. The point here is that no anticipatory fetches are made, even though anticipatory retrieval may occur as a side effect of fetches that are performed in response to explicit retrieval requests by a user. Note that a user can create the effect of anticipatory fetching by simply issuing anticipatory reads for those applications (e.g., monthly payroll) whose future data requests are predictable.

The transfer of data into a level may necessitate the removal of some data already at that level in order to make room for the incoming data. This removal process is referred to as replacement and the replaced data is

said to have overflowed.

The justification of a demand fetch policy is based on the fact that, for hierarchical storage systems, ". . . given any [series of requests] and replacement algorithm (not necessarily using demand paging) [a] replacement algorithm exists that uses demand paging and causes the same or a fewer number of pages to be loaded . . ." [Matt70]. Intuitively, this means that demand paging leads to no more I/O requests within the hierarchy than any other possible algorithm. Since the number of I/O requests is a fundamental limiting factor for system throughput, using a policy which minimizes I/O requests is very attractive.

This policy leads to a retrieval scheme which operates roughly as follows:

1) a user issues a READ request

2) if the requested data *is found in* the cache level it is returned to the requesting user

3) if not found, the hierarchy is searched for the requested data. The requested data is transferred from the level in which it is found to the cache level, and from there to the requesting user.

This very general description leaves open a number of questions which will be addressed in the remainder of this chapter, including:

1) is only the requested data fetched or are entire blocks which may contain data which has not been explicitly requested retrieved?

2) if the data is blocked into pages, should the pages be the same size at all levels?

3) given that an entire page has been referenced, should a copy (or multiple copies) of the page be saved in the levels between the level at which the page was found and the cache level, in anticipation of future requests for data in that page?

4) how is the hierarchy searched - level by level or all levels in parallel?

5) finally, if this strategy results in an overflow, how should the page to be removed be selected, and what should be done with it?

Questions 1, 2, and 3 will be addressed in the next section, while questions 4 and 5 will be dealt with in Section 3.4.

3.3.2 Page Size Specification and Page Splitting

An examination of the random access times and transfer rates of the devices listed in Table 2.1 reveals that access times vary by over six orders of magnitude, while transfer rates vary by only two orders of magnitude. This fact, coupled with spatial locality considerations, can be used to show that system performance can be optimized (with respect to total expected data transfer delay) by

1) a choice of page sizes such that $N^1 < N^2 < \ldots < N^r$ (where N^i is the size of the unit of transfer from Level i+1 to Level i, and also the size of the page stored at Level i) coupled with

2) the use of a page splitting algorithm to determine the placement of data in the various levels in the hierarchy.

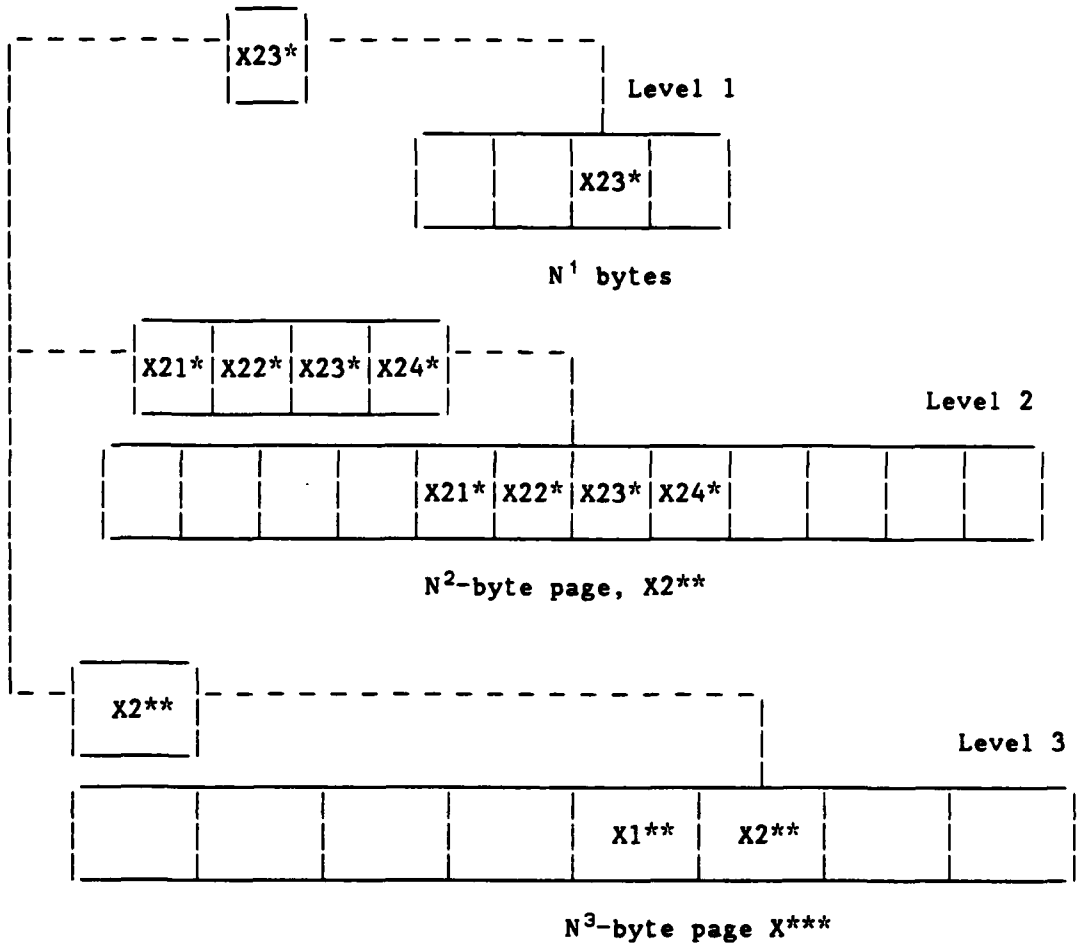
(For a detailed derivation of this result, see [Madn73].) Intuitively, a factor to consider in the choice of N^i is that this choice represents a tradeoff between the sensitivity of the system to spatial and temporal locality, respectively. A smaller page size decreases the sensitivity to spatial locality (less spatially related data is retrieved by each fetch), but increases sensitivity to temporal locality by allowing a level to hold a larger and more diversified collection of pages. Of course, the optimal values of N^i will depend on the actual degree of locality and the speeds of the various storage devices in the hierarchy.

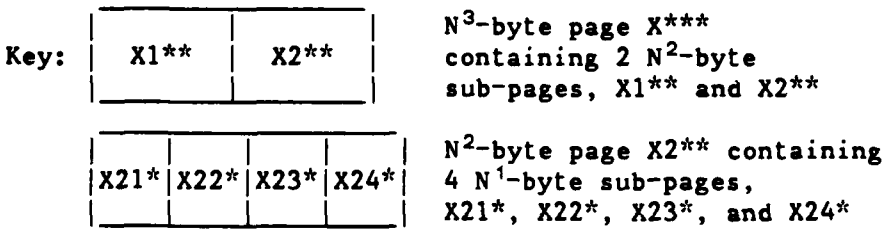
In order to retain as much flexibility as possible in the design of DSH-III, we allow the size of the unit of transfer between Level 1 and the user to differ from the page size, N^1, at Level 1. This idea is consistent

with the variation of page sizes within the storage hierarchy itself. The size of the data blocks transferred, via the UBUS, between DSH-III and a user will be denoted N^0.

Page splitting operates as follows. Suppose a referenced data item is found in some page, of size N^3, at Level 3. The sub-page (of size N^2) containing the referenced data item is retrieved and transferred to Level 2 via the GBUS. Logically, the next step would be for Level 2 to transfer the sub-page of size N^1 containing the referenced data item to Level 1. In practice, this is not necessary, since Level 1 can obtain the appropriate sub-page from the GBUS during the initial transfer of data from Level 3 to Level 2. Therefore, the strategy actually used by DSH-III is for the level at which the data is found to "broadcast" the appropriate page over the GBUS, with each level extracting the appropriate sub-page from the broadcast data. This strategy is called READ-THROUGH since the requested data is read through from the level at which it is found directly to the highest storage level. This procedure is illustrated in Figure 3.2.

This figure gives an example of the notation we will use when it is desired to make explicit the relationships between a set of pages and their sub-pages at various levels. Virtual addresses of the smallest addressable units of data, i.e., pages of size N^0 bytes, will be denoted by sequences of letters and/or numbers. Thus, X, 12345, and AB3 might denote addresses. The addresses of pages at Level i are denoted by a sequence of letters and/or numbers followed by i asterisks. Thus A** denotes the address of a page in Level 2. Page/sub-page relationships are expressed by using identifiers with all non-asterisk symbols equal. Thus A** contains (is the "parent" of) AB* (its "child"). Page AB*, in turn, is the parent of AB3. In situations where the relationships between pages are clear from the





Note: Pages not drawn to scale, i.e., $N^3 > N^2 > N^1$

Figure 3.2 - Illustration of Page Splitting as Page X23* is Broadcast From Level 3 to Levels 1 and 2

context, we will adopt the simpler notation of denoting pages by upper case Latin letters, e.g., X, Y.

In order to simplify implementation of the READ-THROUGH strategy, it is desirable that the page sizes be powers of 2. From now on we will assume that N^i can be written as $2^{k(i)}$, where k(i) is an integer. This means that each page in Level i contains an integral number of sub-pages of size N^{i-1}. The total capacity of Level i, denoted C^i, can be computed as $C^i = m^i N^i$, where m^i is the number of pages of size N^i in Level i.

3.3.3 Page Splitting and Redundant Data

One result of using a page splitting strategy is that the system will contain redundant copies of data blocks. As a data item is read through into Level 1 it leaves a copy of itself, embedded in an appropriately sized page, in each level of the hierarchy.

There are two disadvantages associated with this redundancy. First, from a myopic point of view, redundancy is wasteful of storage space, but this ignores the performance gains resulting from a page splitting policy. One alternative to page splitting would be to transfer the requested data, X say, to Level 1 only (in a page of size N^1). Then, as subsequent references caused other pages to be brought into Level 1, X would eventually overflow and would be moved down to Level 2 to make room for the incoming data. This type of overflow handling policy will be justified in the discussion of overflows in Section 3.4.3. Moving X from Level 1 to Level 2 would result in two I/O's and a bus transfer. In addition, since the pages in Level 1 are N^1 bytes and the pages in Level 2 are $N^2 > N^1$ bytes, the rest of the page of size N^2 containing X will have to be retrieved and moved into Level 2 in order to maintain the consistency of the paging scheme at Level 2. This latter retrieval will result in two more I/O's and another bus transfer. This process would then have to be repeated as the page

removed from Level 2 to make room for X is moved to Level 3, and so on, all the way down the hierarchy. It is clear that this scheme involves considerable overhead, both in terms of extra I/O's and added bus traffic. It will be shown that the redundancy caused by page splitting allows the use of READ and WRITE algorithms which eliminate this overhead by discarding removed pages while still obtaining the performance benefits due to considerations of temporal locality.

The second disadvantage arises out of the potential for inconsistencies present in any system with redundant data. It will be shown that the algorithms used by DSH-III eliminate this possibility.

There are also some positive aspects to the data redundancy in DSH-III. Besides the performance advantages alluded to above, redundancy enhances the ability of DSH-III to recover from failures and continue operation by reconfiguring itself to bypass a failed component.

Finally, note that maintaining redundancy does not have much impact on the total effective storage capacity of the hierarchy since, in any reasonable design, one would expect $C^i << C^{i+1}$.

The advantages of using a page splitting policy appear to outweigh the disadvantages. Therefore, this policy has been incorporated into the data movement algorithms of DSH-III.

3.4 READ Strategies

Up to this point we have discussed data movement strategies and the tradeoffs among various alternative policies at a fairly general level. After settling on a hierarchical structure and specifying possible data movement paths between levels of the system, various data management policies and tradeoffs were discussed. The strategies selected on the basis

of this discussion included demand fetch with replacement, different page sizes at various levels (selected on the basis of storage device speeds and locality considerations), page splitting, and the storage of redundant data. The following sections further refine the specification of the READ algorithms used by DSH-III. Issues addressed include specification of how the READ-THROUGH policy operates, how pages are selected for replacement, and how overflows are handled.

3.4.1 Data Location and READ-THROUGH

If a referenced data item, X say, can not be located in Level 1 (the cache level), the rest of the hierarchy must be searched for X. There are two ways that this search might be implemented, serial or parallel. A serial search could operate as follows. The READ request is passed, via the GBUS, from Level 1 to Level 2. A directory in Level 2 is searched for a page containing X. If found, X is retrieved from the storage system in Level 2, and the READ-THROUGH process is initiated to transfer X to Level 1 and the Functional Hierarchy. If X is not found in Level 2, the READ request is passed down to Level 3, and the search is repeated at that level, and so on, until the request has percolated down to a level which contains X.

A parallel search operates as follows. The READ request is broadcast from Level 1 to all lower levels of the hierarchy, which then perform simultaneous, parallel directory searches for X. The highest level to locate X then initiates the READ-THROUGH for X, after informing all the other levels that X has been located.

The parallel search scheme has an obvious advantage in that it has potential for minimizing the expected delay between initiating the search and locating the data. On the other hand, there are some potential drawbacks to the parallel search strategy. While it is true that the expected time for a parallel search is less than that for a serial search, the parallel search is wasteful of system resources in that only the results of one search will be used. All the other searches represent wasted effort. This would not be a concern if it was expected that there would be excess processing power at each level. However, the fundamental rationale for the multi-processor architecture of DSH-III is that throughput can be increased by processing multiple transactions in parallel. Thus the parallel search might use resources that would otherwise be employed doing "useful" work as a part of the inherent parallelism of DSH-III. There is an implicit assumption here that the search uses scarce system resources. If, for example, the search was performed by specialized (and underutilized) hardware, there would be no disadvantage to parallel searches. Intuitively, the parallel search strategy represents an attempt to decrease response time for individual transactions at the potential expense of decreasing overall system throughput. But note that the time for a serial search grows linearly with the number of levels that must be searched, while the data retrieval times grow by orders of magnitude as one moves down the hierarchy. This implies that the parallel search strategy is relatively less advantageous for retrieving data from lower levels of the hierarchy because the total search time for either strategy is relatively insignificant compared to the data retrieval time at lower levels. Therefore, for those retrievals where the parallel strategy has the greatest absolute advantage, the relative impact on response time is small. In addition, locality

considerations lead one to expect that the majority of retrievals will be satisfied at high levels of the hierarchy. For retrievals from these levels, the serial search time is not much greater than the parallel search time, and thus does not have a great impact on response time.

Another drawback of the parallel search strategy is the extra algorithm complexity and control protocol overhead needed to coordinate the searches at different levels in order to determine the highest level at which the data was found.

The tradeoffs between serial and parallel searching will be the subject of further investigation, since the relative merits of the two schemes are dependent on the exact hardware configuration and load on the various components of the system. For the purposes of this paper, we will use the serial search strategy because it is simpler and because the parallel strategy does not seem to offer any significant performance benefits.

The precise mechanics of the READ-THROUGH process are fairly straightforward. Once the requested page has been located, it is retrieved from the LSS and broadcast to all higher levels of the hierarchy.

One potential way to decrease system response time would be to order broadcasts so that data destined for higher levels was broadcast first. In particular, the page destined for the cache level (and the Functional Hierarchy) could be broadcast first. In other words, the first data broadcast would be the sub-page of size N^1 containing the referenced virtual address. Next, the sub-page of size N^2 containing the referenced data would be broadcast, and so on until the entire data block had been broadcast. This refinement of the basic READ-THROUGH strategy is not considered further in this paper, but will be a subject for further investigation.

3.4.2 LRU Replacement

The next issues to be addressed deal with the choice of a replacement policy and the design of overflow handling algorithms.

DSH-III will use a Least Recently Used (LRU) replacement policy. At any point in time, the pages at a level can be ordered by time of most recent reference. This ordering is called the LRU stack for the level, and each page at the level occupies a unique position in the stack (which will change as references are made to the level). Under the LRU policy, the page at the bottom of the stack (i.e., the page least recently referenced) is the one selected for removal. There are three reasons for the selection of LRU as the replacement algorithm for DSH-III:

1) LRU has been shown empirically to compare favorably with the "optimal" removal algorithm, MIN [Bela66]. (MIN itself can not be used as a replacement algorithm since it requires knowledge of the future.)

2) LRU is one example of a class of "stack algorithms" [Matt70]. These algorithms can be shown to have "inclusion" properties which are used to simplify the data movement algorithms of DSH-III.

3) One consequence of the inclusion property of LRU is that the "hit ratio" for a level (the fraction of READ requests satisfiable at that level) is a monotonic function of level size. Thus LRU is not subject to certain anomalies [Bela69] which can decrease performance as level size increases.

Now suppose that a reference to some data item, X, is satisfied at some level, j (the highest level containing X). This will cause X to be moved to the top of the LRU stack at Level j. Then the READ-THROUGH for X will result in a copy of X being stored in all levels, i, with $i < j$, and all these levels will have their LRU stacks changed to reflect the fact that the most recent reference at each of these levels was a reference to X. The algorithm described up to this point is referred to as LOCAL-LRU [Lam79a]. It is characterized by LRU stack updates being performed only at those

levels which actively participate in a READ-THROUGH, in this case Level 1 to Level j.

An alternative policy is termed GLOBAL-LRU. Under this policy, the LRU stacks in Level 1 to Level j would be updated as for LOCAL-LRU. In addition, the LRU stacks in all levels below Level j which contained X would also be updated just as if a reference to X had been made at each of those levels.

Figures 3.3 and 3.4 illustrate these two alternative LRU policies.

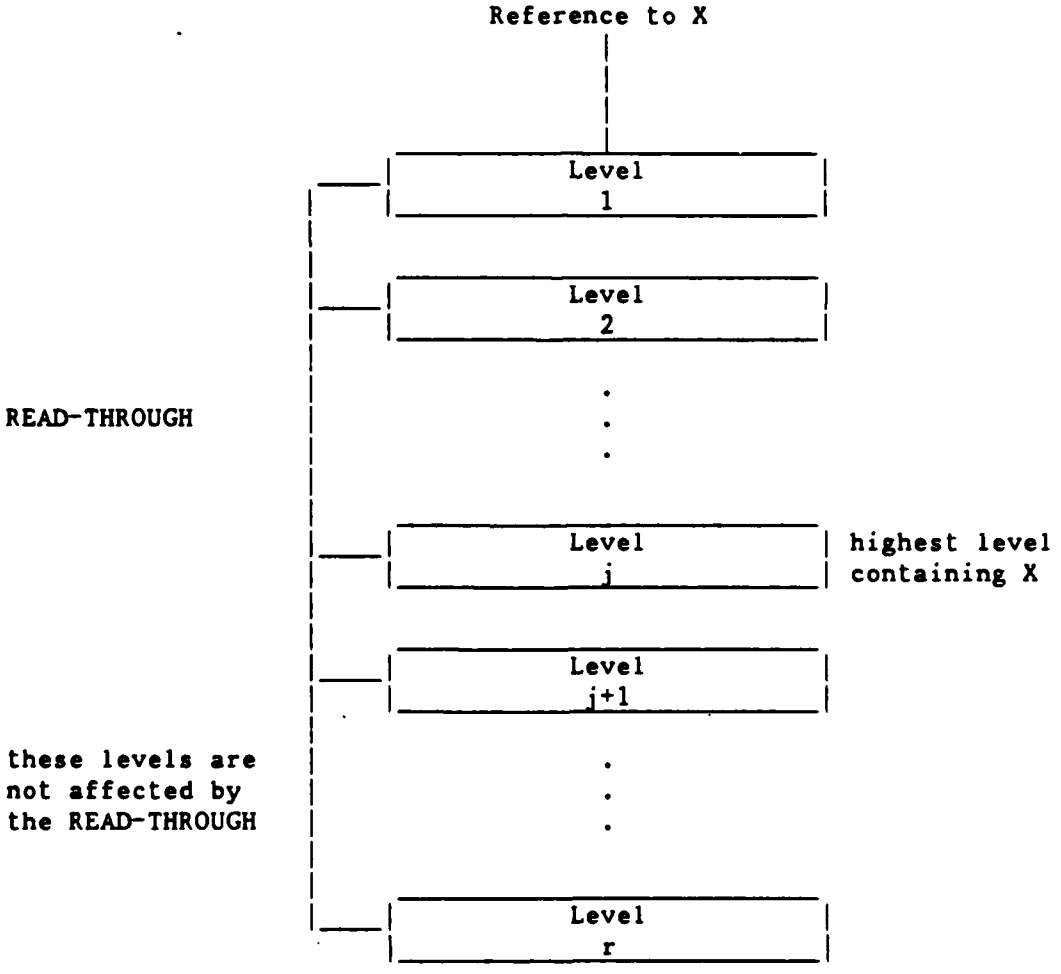


Figure 3.3 - Illustration of LOCAL-LRU Algorithm

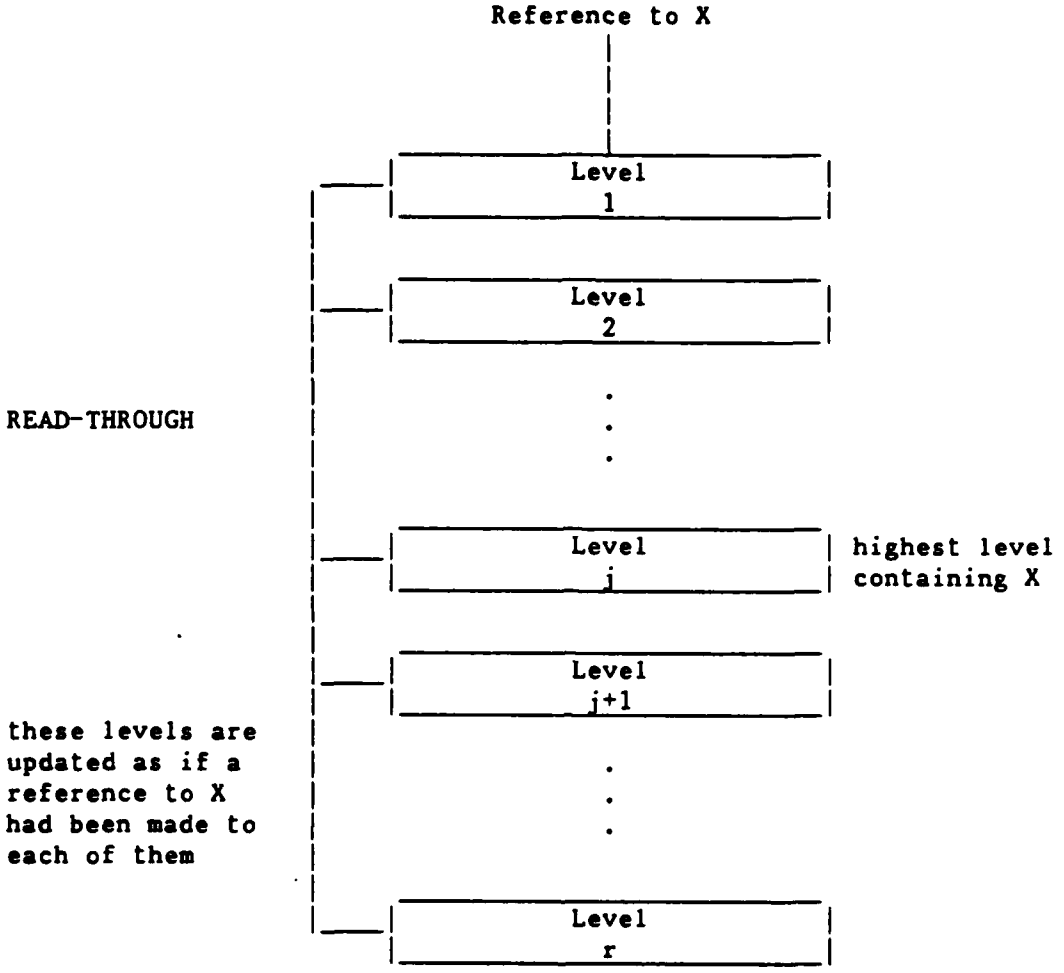


Figure 3.4 - Illustration of GLOBAL-LRU Algorithm

The justification for the choice of LRU policy used by DSH-III will be deferred until the issue of overflow handling has been discussed.

3.4.3 Overflow Handling

When an overflow occurs there are three possible courses of action:

option 1: The overflow page could be simply discarded. (Since a copy of every page exists in the reservoir, this policy does not lead to the loss of any data.)

option 2: The overflow page could be moved to the next lower level of the hierarchy.

option 3: The overflow page could be moved to some other location(s) in the hierarchy.

Before presenting the pros and cons of these alternatives, let us recall the original rationale for adopting a hierarchical system design. The idea was to create a structure which would take advantage of locality by providing a range of storage devices and allowing the dynamic allocation of data to faster or slower devices based on anticipated future usage patterns. Therefore, we adopt as a "preferred" overflow policy one which attempts to keep more recently referenced data in higher (and faster) storage levels. The obvious way to accomplish this is to maintain an LRU stack for the entire data base, and to allocate pages to higher or lower levels according to their position in this global *LRU stack*. Clearly, this is infeasible, since a global LRU stack runs contrary to the principle of distributed control and hierarchical decomposition. Instead, we can approximate the ideal strategy by maintaining separate LRU stacks within each level, and moving pages down the hierarchy one level at a time each time they overflow.

This implies that option 2 should dominate option 3, so option 3 is discarded as a potential overflow handling strategy. Thus the choice of overflow policy is reduced to selecting either option 1 or option 2 on the basis of degree of conformance with the "preferred" policy (subject, of course, to performance considerations).

In order to define option 2 completely, we must specify what effect an overflow from Level i has on the LRU stack at Level i+1. Lam [Lam79b] proposes two algorithms for handling this situation: Static Overflow Placement (SOP) and Dynamic Overflow Placement (DOP). Under an SOP policy,

the overflow of a page, X, from Level i has no effect on Level i+1 unless there is no copy of X at Level i+1, in which case the overflow is treated as if a reference to X had been made at Level i+1. Under DOP, an overflow of a page, X, from Level i results in the LRU stack at Level i+1 being updated as if a reference to X had been made at Level i+1.

Two things should be noted regarding these policies. First, under either SOP or DOP, if X is not in Level i+1, the reference generated by the overflow results in a READ-THROUGH of X to Level i+1. Therefore, in this case SOP and DOP have identical effects but involve significant overhead. Second, if X is found in Level i+1, SOP implies that no action need be taken beyond verifying that X is indeed in Level i+1. Notice that in this case (option 2 with SOP and X found in Level i+1), option 1 and option 2 are equivalent, except for the verification that X is in Level i+1.

Tables 3.1a and 3.1b summarize the pros and cons of the three overflow handling options in the light of the foregoing discussion. Table 3.1a assumes that there is no redundant data in the system, while Table 3.1b assumes that the parent page of any overflow page will always be in the next lower level. We term this property Overflow Inclusion. This distinction is important because, as can be seen from Tables 3.1a and 3.1b, Overflow Inclusion eliminates the need to perform any actual data transfers in order to support any of the three overflow handling options. With Overflow Inclusion, the only difference between the three options is the need, under option 2 with DOP, to update the LRU stack at Level i+1 in the event of an overflow from Level i. Notice that the assumption of Overflow Inclusion eliminates the need, under SOP, to determine whether or not an overflow page has a copy in the next lower level.

	Option 1	Option 2 (SOP)	Option 2 (DOP)
Conformance with Preferred Policy	low	high	high
Bus Load	low	high	high
Complexity	low	moderate	moderate

Table 3.1a - Overflow Policy Tradeoffs (No Redundancy)

	Option 1 and Option 2 (SOP)	Option 2 (DOP)
Conformance with Preferred Policy	high	high
Bus Load	none	low
Complexity	low	moderate

Table 3.1b - Overflow Policy Tradeoffs (Overflow Inclusion)

Tables 3.1a and 3.1b show that the "best" overflow handling scheme can be achieved in a system with Overflow Inclusion. Without Overflow Inclusion, the choice is between a policy (option 1) which is simple and involves little overhead but does not have the desirable features of the "preferred" policy and a policy (option 2) which conforms to the "preferred" policy but imposes heavy overhead on the system. The next section shows how Overflow Inclusion can be achieved at little cost in terms of overhead or complexity by simply placing loose bounds on m^1.

The original rationale for DOP (as opposed to SOP) was that DOP conformed more closely with the "preferred" overflow concept, in that an overflow from Level i would move from the last slot in the LRU stack at Level i to the first slot of the LRU stack at Level i+1. This agrees with the view of the stack at Level i+1 being a logical extension of the stack at

Level i. A closer examination reveals that this advantage of DOP over SOP is largely illusory, if GLOBAL-LRU is used. Assuming that there is redundancy in the system, the effect of DOP is to move X, the overflow from Level i, ahead (in terms of LRU stack position in Level i+1) of all the pages in Level i+1 which contain sub-pages in Level i. Thus the page, X, which is no longer in Level i, is, in some sense, being treated as if it were more recently referenced than pages which are still in Level i. Of course, when these latter pages eventually overflow from Level i, the DOP algorithm will restore them to their "rightful" place ahead of X in the LRU stack in Level i+1. However, Theorem 4 in the following section will show that SOP, in conjunction with GLOBAL-LRU, has approximately the same end result, with none of the complexity or overhead of DOP. Since we will be using an algorithm based on GLOBAL-LRU, SOP appears preferable to DOP as an overflow policy.

3.4.4 Multi-Level Inclusion (MLI) and Overflow Inclusion (MLOI)

If, at any instant of time, any page X in Level i is a subpage of some page in Level i+1, the storage hierarchy is said to have the Multi-Level Inclusion (MLI) property [Lam79a]. At first glance, it might seem that MLI is sufficient to guarantee the Overflow Inclusion mentioned in the preceding section, but this turns out to be false. Consider the three level hierarchy shown in Figure 3.5a. In this hierarchy, the same data, X, is in the next page to be selected for eviction from both Level 1 and Level 2. Figure 3.5b shows the effect of a READ-THROUGH from the reservoir in this situation. As a result of the READ-THROUGH, Level 1 and Level 2 are updated simultaneously, resulting in a simultaneous overflow of X from each level. Thus at the instant that X overflows from Level 1, there is no copy of X in

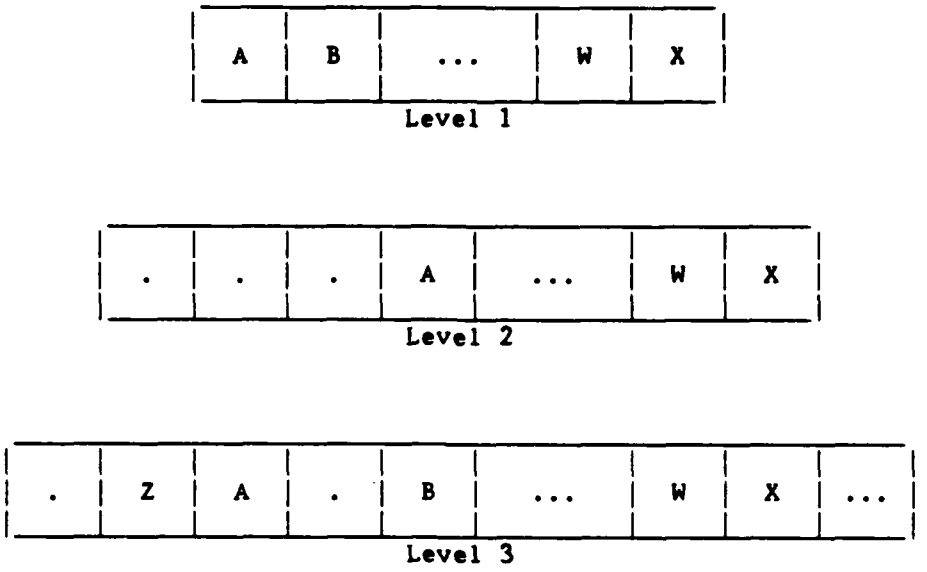


Figure 3.5a - Hierarchy Just Before READ-THROUGH for Z

Level 2, even though the MLI property holds for this hierarchy. In this case, discarding the overflow page would leave no copy of X in Level 2, an undesirable outcome from the point of view of taking full advantage of temporal locality. An inclusion property slightly stronger than MLI is needed to prevent this situation from occurring. This property is Multi-Level Overflow Inclusion (MLOI). MLOI holds if any overflow page from Level i is a subpage of some page in Level i+1 and MLI holds as well. Thus, MLOI is sufficient to allow overflows to be discarded under any reasonable overflow handling policy. The following sections describe the conditions under which the MLI and MLOI properties can be guaranteed to hold, and briefly discuss the implications of these properties for performance and reliability of DSH-III.

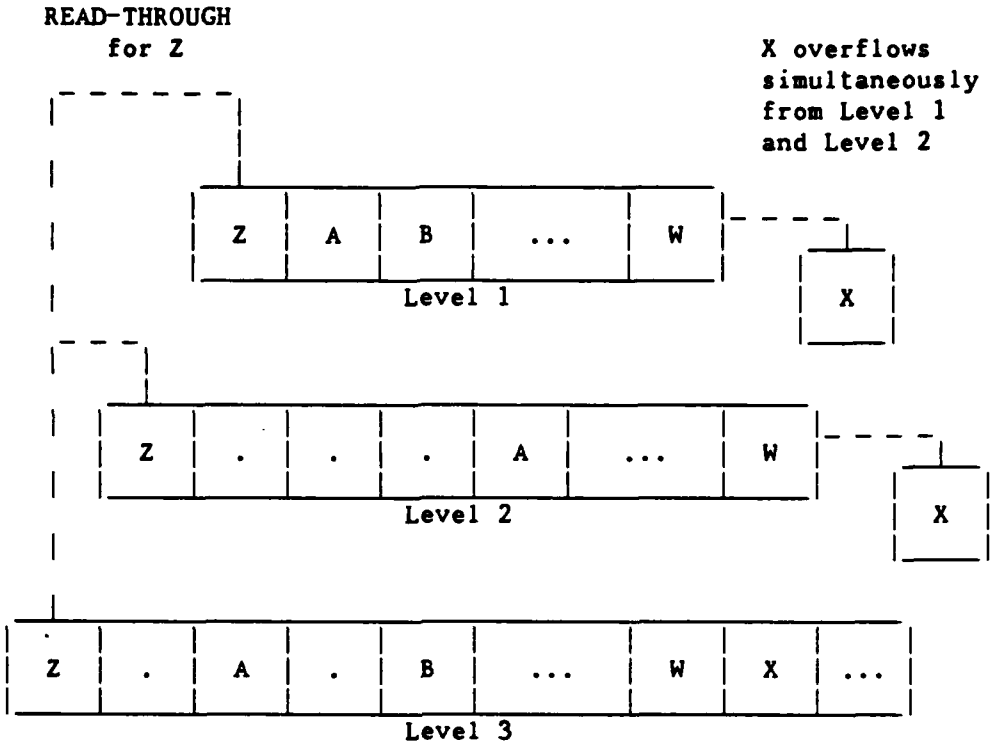


Figure 3.5b - Hierarchy Just After READ-THROUGH for Z

3.4.4.1 Theoretical Basis for MLI and MLOI

In order to define precisely the conditions under which MLI and MLOI hold it is necessary to completely specify the data movement algorithms used by DSH-III. The assumptions that have been made so far regarding what would constitute a "good" algorithm are:

- the use of READ-THROUGH with page splitting
- the use of an LRU replacement policy at each level
- the "preferred" overflow handling policy involves logically moving an overflow page to the next lower level. Recall that if MLOI holds, this policy imposes no overhead on the system, since no actual data movement is needed.

Given these basic assumptions, there are four possible READ strategies that can be derived by selecting either LOCAL- or GLOBAL-LRU in combination with either Static Overflow Placement (SOP) or Dynamic Overflow Placement (DOP). Table 3.2 shows the four possible strategies.

		Overflow Handling Policy	
		Static Overflow Placement (SOP)	Dynamic Overflow Placement (DOP)
LRU Policy	LOCAL-LRU	LOCAL-LRU-SOP	LOCAL-LRU-DOP
	GLOBAL-LRU	GLOBAL-LRU-SOP	GLOBAL-LRU-DOP

Table 3.2 - The Four Possible READ Algorithms

We now present a series of theorems which characterize the inclusion properties of hierarchical storage systems using these four algorithms [Lam79b].

Theorem 1: Using any of the four algorithms, if $m^1 \geq 2$ and $m^j \leq m^{j-1}$ for some j then there exists a reference string which leaves the system in a state where MLI does not hold.

This theorem implies that, if MLI is to be guaranteed, m^j must be strictly greater than m^{j-1} for all j. In other words, the number of pages in each level must be greater than the number of pages in the next higher level. This is not a restrictive condition since one of the precepts of the hierarchical design is to have the capacities of the levels increase from top to bottom of the hierarchy.

Theorem 1 gives necessary conditions for MLI (and therefore MLOI) to hold, but these conditions are not sufficient, as is shown by the next theorem.

Theorem 2: Using LOCAL-LRU-SOP or LOCAL-LRU-DOP, if $m^1 \geq 2$, there exists a reference string which leaves the system in a state where MLI does not hold.

Based on this theorem, we reject the two algorithms using LOCAL-LRU as potential candidates for DSH-III.

The next theorem gives conditions on m^i which guarantee that MLOI (and therefore MLI) holds for all possible reference strings for the two algorithms using GLOBAL-LRU.

Theorem 3: Using GLOBAL-LRU-SOP, if $m^1 \geq 2$, MLOI holds for any reference string if and only if $m^j > m^{j-1}$. Using GLOBAL-LRU-DOP, if $m^1 \geq 2$, MLOI holds for any reference string if and only if $m^j \geq 2m^{j-1}$.

This theorem gives fairly loose conditions (especially for GLOBAL-LRU-SOP) on the relative sizes of the levels of a hierarchy which are sufficient to guarantee that MLOI always holds.

Thus if a hierarchy is subject to the constraints of Theorem 3, we can implement a READ strategy based on either GLOBAL-LRU-SOP or GLOBAL-LRU-DOP, which constitutes a "good" algorithm based on the arguments presented so far in this paper.

3.4.4.2 Performance Implications of MLI and MLOI

The performance benefits of a policy which attempts to maintain MLI and MLOI have been discussed above, and are briefly summarized here. They include

- support for a "preferred" overflow policy with no attendant data transfer overhead

- conformance with the principle of using varying page sizes in conjunction with page splitting in order to minimize expected retrieval times

- enhancement of system availability by allowing intentional (e.g., for preventive maintenance) or unintentional (e.g., in case of failure) removal of a level without changing the logical structure of the system or its data movement algorithms.

The drawbacks associated with maintaining MLI and MLOI arise from two sources

- extra complexity and processing overhead within each level

- extra interlevel communication overhead

Each of these sources will contribute to a degradation in performance in varying degrees, depending on the exact policy used to maintain MLI and MLOI.

As an example of one such policy, the reader is referred to Lam's approach [Lam79a], which presents a set of READ and WRITE algorithms which maintain MLI. This approach is based on the idea of associating with each page at each level a USC (upper storage copy) flag which indicates whether or not a sub-page of the page is resident in some higher storage level. Lam uses a modified LRU replacement policy which selects the least recently used page which does not have its USC flag set as the candidate for replacement.

In order to maintain the correct USC flag values, a level must notify the next lower level whenever it evicts the last sub-page of some page, X. This policy is complicated by the possibility that the notification of the eviction of the last sub-page of X could occur simultaneously with a READ-THROUGH for some other sub-page of X. This situation is called "erroneous overflow". Besides imposing a significant computational burden, Lam's method of handling erroneous overflow makes unrealistic assumptions

about the hardware, i.e., that a message can be purged after it has been sent, but before it has been received. Obviously, supporting this capability involves very delicate and complex interactions between levels, and enormously complicates the interlevel communication protocols. In addition, there are a number of other subtle pathological cases such as "racing requests", and "overflows to partially assembled blocks" that must be dealt with.

While Lam's technique for maintaining MLOI has the advantage of being conceptually simple (i.e., it is intuitively obvious that this algorithm does, in fact, work), the intricate and computationally burdensome overflow handling would render it impractical even if the required hardware could be provided.

The algorithms proposed in this paper attempt to maintain MLOI by implementing GLOBAL-LRU-SOP subject to the conditions of Theorem 3, above. In essence, MLOI is obtained as a by product of the data movement algorithm, and therefore we do not need to consider any of the pathological cases that greatly increase the complexity of Lam's approach. The advantages of the approach presented herein include

- no overhead for overflow handling
- avoidance of much of the computational complexity implied by Lam's algorithms.

On the other hand, this approach has some disadvantages, including

- a need to perform strict LRU replacement
- a need to synchronize LRU updates between levels (in order to perform GLOBAL-LRU properly)
- restrictions on the relative sizes of the levels of the hierarchy, as specified by Theorem 3 (although, as noted before, these restrictions are not constraining on any reasonable design, that is, one for which the number of pages per level increases from Level i to i+1). Furthermore, this restriction is implicit in Lam's algorithms since obviously MLI can not be

maintained if $m^j < m^{j-1}$ for any j.

Table 3.3 summarizes the comparison between Lam's READ algorithms and the ones proposed in this paper.

	Lam's Algorithms	Proposed Algorithms
Replacement Policy Restrictiveness	unrestrictive	fairly restrictive
Replacement Policy Complexity	fairly high	fairly low
Overflow Handling Overhead	moderate	none
Constraints on Hierarchy Structure	loose	loose

Table 3.3 - Comparison of Two Strategies for Maintaining MLOI

The only significant advantage of Lam's algorithms appears to be in the first category, replacement policy restrictiveness, while its only significant disadvantage is in the second category, replacement policy complexity. Thus a choice between these two policies will turn on which of the two categories has the greatest impact on performance. In point of fact, Lam's simulation studies [Lam79a] showed that performance is limited by bus bottlenecks, rather than processing bottlenecks, although these studies appear to have been based on optimistic estimates of 1985 micro-processor technology. Both Lam's algorithms and those proposed herein represent an attempt to lower bus contention (by reducing the page transfers needed to support overflow handling) at the expense of increased processing overhead and complexity. In light of Lam's simulation results, this general approach appears to be a plausible method of reducing the bus bottlenecks in

the system. With this in mind, the proposed algorithms, based on GLOBAL-LRU-SOP, would seem to have an overall advantage, due to their lower bus utilization and less complex replacement policy. A final choice between the two policies will depend on the results of emulation studies to be performed using a proposed Software Test Vehicle (STV) [To81]. On balance, however, a set of algorithms based on GLOBAL-LRU-SOP would seem to have almost all of the properties desirable in a data movement strategy, while still being quite simple and efficient.

3.4.5 Implementation Issues for GLOBAL-LRU-SOP

This section discusses a number of issues relevant to the implementation of a READ algorithm based on GLOBAL-LRU-SOP. For the sake of brevity, from now on this algorithm will be denoted GLS. The issues to be discussed are

1) pre-eviction of pages,
2) LRU update epoch selection,
3) LRU update synchronization, and
4) duplicate READ request handling.

A future paper will present the details of an efficient software implementation of an LRU eviction algorithm.

3.4.5.1 Pre-eviction of Pages

Pre-eviction of pages refers to the process of selecting pages for replacement before they are explicitly forced out of a level by a READ-THROUGH (compare with post-purge used by Multics [Orga72]). Of course, MLOI will be destroyed if a page with a sub-page in the next higher level is evicted in this manner. A pre-eviction algorithm which does preserve MLOI

can be deduced from the following theorem.

Theorem 4: Define $S^j(X)$ to be the LRU stack position of page X in the LRU stack at Level j. Thus $S^j(X) = 1$ for the most recently referenced page, and $S^j(X) = m^j$ for the least recently referenced page in Level j. Using GLOBAL-LRU-SOP, if $m^1 \geq 2$ and $m^j > m^{j-1}$, then for any page X in Level j and sub-page, Y, of X in Level j-1, $S^{j-1}(Y) \leq k \Rightarrow S^j(X) \leq k$.

Proof: First note that the statement of the theorem can be written as $S^j(X) \leq S^{j-1}(Y)$. We will show that the theorem is true immediately after any reference to Y, and that succeeding references do not change the inequality. Immediately after Y is referenced, we have $S^j(X) = S^{j-1}(Y) = 1$. A succeeding reference either touches X, or it does not. If it does not reference X, then as a result of the reference both $S^j(X)$ and $S^{j-1}(Y)$ increase by one. If it does reference X, but does not reference Y, then, as a result of the reference, $S^j(X) = 1$ and $S^{j-1}(Y)$ increases by one. In either case, the inequality holds. QED

What this theorem says is that a page is always closer to the top of the stack than any of its children in higher levels. Based on this theorem, a page, X, can be safely pre-evicted from Level j as long as $S^j(X) \geq m^{j-1} + 1$. Intuitively, what is going on is that pre-eviction is reducing the effective size of Level j, and as long as the effective size of Level j is not reduced below the limit specified by Theorem 3, MLOI will be preserved. This ability to dynamically alter the effective size of a level is a reflection of the "stack inclusion" property of stack algorithms, such as LRU [Matt70]. Note that the criteria for pre-eviction at Level j, as specified by Theorem 4, depend only on the stack at Level j and the number of pages in

Level j-1. Thus the pre-eviction algorithm does not depend on any dynamically changing information which is not local to Level j.

In practice, it will turn out that some pages in a level may be locked against eviction. This would be the case if, for example, a page was in the process of being retrieved in preparation for a READ-THROUGH, but the LRU update for the page had not yet been performed. In this case, it is not possible to evict the locked page. In order to prevent this situation from degrading performance, the pre-eviction algorithm does allow some deviation from strict LRU eviction by skipping over locked pages. Once again, appeal to the stack inclusion property of LRU shows that this process does not violate Theorem 3, and thus preserves MLOI.

One problem with pre-eviction is that if all pages, X, with $m^{j-1} < S^j(X) \leq m^j$ are locked, then there are no candidates for pre-eviction at Level j. Thus, Level j could run out of available page frames, and all READ-THROUGH's from below Level j would be blocked.

To see that this blocking phenomenon can not cause a deadlock, note that a level can only block levels below it. Therefore, Level 1 can never be blocked in this fashion, and can always accept a READ-THROUGH (after waiting for a previous READ being satisfied at Level 1 to complete, if necessary). Thus, Level 2 can not be involved in a deadlock because it can always eventually initiate a READ-THROUGH to Level 1, thus freeing a page frame in Level 2. Similarly, this implies that Level 3 can never be involved in a deadlock, and so on for the rest of the levels in the hierarchy.

One final problem with pre-eviction is that, by reducing the effective size of a level, it degrades system performance. Thus, the ideal pre-eviction algorithm should strike a balance between not pre-evicting

enough pages (thus potentially blocking READ-THROUGHs temporarily), and pre-evicting too many pages (thus reducing performance by reducing available storage at some level).

3.4.5.2 LRU Update Epoch Selection

It is possible that some time might elapse between the initial READ request and the reflection of the reference in the LRU stacks at the various levels. The GLS algorithm attempts to perform the LRU update as closely as possible in time to the point at which the actual READ-THROUGH of data to Level 1 is performed. This policy has been adopted so that the LRU stacks reflect as accurately as possible the sequence in which references are satisfied, rather than the sequence in which they are initiated. This policy seems to adhere most closely to the spirit of LRU replacement, and minimizes anomalies wherein a page has become a candidate for eviction from Level 1 before it has actually been retrieved. This would clearly be an undesirable situation.

3.4.5.3 LRU Update Synchronization

GLOBAL-LRU requires "simultaneous" LRU stack updates at each of the levels. It is not necessary that an LRU update be processed at the same time at each level. All that is required is that LRU updates be processed in the same order at each level. This can be accomplished by merely processing LRU updates in FIFO order at each level. These updates will be accomplished by broadcasting an LRU update message to all levels just prior to initiation of a READ-THROUGH. The READ-THROUGH itself can not be used as a synchronization signal because that would require that a READ-THROUGH be sent to all levels, not just higher levels.

3.4.5.4 Duplicate READ Request Handling

The final GLS implementation issue is duplicate READ request handling. Suppose two READ requests, for pages X1* and X2*, are received simultaneously, and both X1* and X2* are sub-pages of the same page, X**, of size N^2. Also suppose that X** is not in Level 2 (implying that neither X1* nor X2* are in Level 1, by MLI). Without loss of generality, assume that the request for X1* reaches Level 2 before the request for X2*. Then, when the request for X2* reaches Level 2, that level will already be expecting X** to be read through to Level 2, in order to satisfy the reference to X1*. We say that X** is "pending" at Level 2. Instead of forwarding the request for X2* to Level 3, Level 2 can hold the request until X** is transferred into Level 2, and then continue processing the request for X2* as if X** had been in Level 2 all along. This policy has been adopted in the expectation that the processing overhead involved in keeping track of pending requests will degrade performance less than processing duplicate requests (such as the one for X**) independently, and thus incurring duplication of effort in the retrieval of X**. This policy also avoids the complication of having X** read through to Level 2 (as a result of the request for X2*) and finding a copy of X** already in Level 2 (as a result of the request for X1*).

3.5 WRITE Strategies

We now turn to a discussion of the WRITE strategies employed by DSH-III. It turns out that many of the problems encountered in designing suitable WRITE algorithms have already been addressed in the development of the READ algorithms for DSH-III. Indeed, many of the design issues mentioned previously were included in anticipation of their relevance to specification of WRITE strategies for DSH-III. Therefore, the brevity of

this section is not an indication that WRITE is simpler than READ, but rather reflects the fact that READ and WRITE strategies have many common issues and problems which have already been dealt with. With this in mind, this section will concentrate on those issues particularly relevant to developing a WRITE strategy for a hierarchical storage system. These issues include reliability, update consistency, and buffer management. As always, the emphasis will be on maximizing performance, subject to constraints imposed by considerations of reliability, maintainability, and cost.

The WRITE process is initiated by a user issuing the command

WRITE(request_id,virtual_address,data).

This command is passed, via the UBUS, to Level 1 of DSH-III. At this point, Level 1 takes a number of actions in order to process the WRITE. These actions include

- making duplicate copies of the updated data in local memory at Level 1 in order to increase availability, i.e., in order to decrease the chance of losing an update if there is a (non-fatal) failure within Level 1,
- sending a "WRITE-complete" acknowledgement back to the user,
- ensuring that the N^1-byte page containing the virtual address to be updated is present in Level 1,
- possibly combining the update with other updates for the same page in Level 1, and
- initiating the process of transferring the update from Level 1 to the reservoir.

The remainder of this section will enlarge upon these points and justify the data management policies implied by them.

3.5.1 Initial Level 1 WRITE Processing

In order to guard against the possibility of lost updates, Level 1 will make a duplicate copy of every update. The copies of each update will be kept in independent memory modules at Level 1, thus providing protection against lost updates in the case of a memory failure at Level 1. Only after the update has been duplicated will a WRITE-complete acknowledgement be returned to the user. Thus, from a user's point of view, a WRITE will complete almost as fast as a READ which is satisfied at Level 1. The process of making a permanent copy of the update in the reservoir can now proceed asynchronously with the handling of subsequent user requests.

The first step in this process is to ensure that the N^1-byte page containing the virtual address to be updated is present in Level 1. This is done by issuing a READ for that page, if it is not in Level 1. In most cases, however, the user will have read the page, preparatory to modifying it, and the page will already be in Level 1.

The second step in this process is to transfer the updated N^1-byte page from Level 1 to the reservoir.

3.5.2 Alternative Store Policies

We refer to the process of transferring an updated page from Level 1 to the reservoir as a Store process. There are four plausible Store policies which will be presented here. They are

- Store Through,
- Store Replacement,
- Store Behind, and
- Staged Store Through.

The function of any of these four policies is to transfer an updated N^1 byte page from Level 1 to Level r (or, more precisely, to an input buffer in Level r).

Before discussing the pros and cons of these policies, we introduce the concept of "coalescing" of updates. Suppose X* has been updated by the request

WRITE(request_id1,X1,new_value_of_X1)

Further suppose that the request

WRITE(request_id2,X2,new_value_for_X2)

arrives after the previous update to X* but before the Store process for X* has been initiated. Now, rather than initiating two Store processes for X*, the two updates, to sub-pages X1 and X2 of X*, can be coalesced and only one (twice-updated) copy of X* need be transferred to Level r. The coalescing of updates reduces the number of Store actions needed in this case by a factor of two.

Now consider a series of WRITE requests. In the short term, these requests can complete at a rate which is dependent mainly on the speed of Level 1. In this case, one can view Level 1 as a buffer between the user and the reservoir (which is the final repository for all updates). In the long run, however, the average throughput for WRITEs is limited by the speed of the I/O devices in the reservoir. The effects of this limitation can be mitigated in either of two ways:

1) reducing the number of updates reaching the reservoir, and
2) increasing (in some unspecified way) the effective size of the buffer between the user and the reservoir.

Point 2 has the effect of making the system less sensitive to transient peaks in the arrival rate of WRITEs, but does not address the fundamental limitation on the average arrival rate of WRITEs imposed by the long term

rate at which the reservoir can absorb the updates. Point 1 does address this issue. For example, suppose k WRITEs per second is the highest rate which can be supported without coalescing. Then a coalescing algorithm which, on average, combined two WRITEs, as in the above example, would allow the system to support, in the long run, a gross WRITE arrival rate of 2k per second. The arrival rate of updates to the reservoir would be half this rate, or k per second, which, by assumption, is within the processing capacity of Level r. In general, if the system can coalesce an average of c WRITEs into each Store action, it will be able to support a gross arrival rate of WRITEs up to c times higher than the rate which could be supported with no coalescing. Since the ratio of READs to WRITEs is fixed for any application, this implies that a coalescing strategy can have a significant impact on overall system throughput. Of course, the overall improvement in throughput would be less than the factor of c since coalescing does not effect the rate at which READs can be processed.

A final point to be made regarding coalescing is that the degree of coalescing attainable (i.e., the value of c) is dependent on two factors:

1) the amount of time that elapses between the receipt of an update and the initiation of the Store for that update, and

2) the degree of WRITE locality exhibited by the system.

These two factors will play an important role in the selection of a suitable Store policy for DSH-III.

We now turn to a discussion of the advantages and disadvantages of the four Store policies.

3.5.2.1 Store Through

Under a Store Through policy, as each update is received, Level 1 broadcasts it to all the lower levels. Under this policy, there is no opportunity for coalescing at Level 1 and there is no flexibility in the choice of epoch at which the Store is performed. On the other hand, Store Through is inherently reliable, since the update is reflected to all levels immediately. Under this policy, we could dispense with duplicating the update at Level 1, as long as the WRITE complete acknowledgement was delayed until after the update had been broadcast.

Finally, note that each update reaches the reservoir as part of an N^1-byte block.

3.5.2.2 Store Replacement

Under this policy, an updated page is held in Level 1 until it is selected for replacement by the LRU replacement algorithm. It is then moved to Level 2, where it is held until selected for replacement, at which time it is moved to Level 3, and so on. As with Store Through, this policy completely restricts the choice of epoch at which the Store is performed, and has the added drawback of imposing a delay on each eviction operation. On the other hand, Store Replacement does provide a maximal "window" during which coalescing can take place, and, in fact, allows coalescing at each level as the update is moved down the hierarchy.

3.5.2.3 Store Behind

A Store Behind policy attempts to alleviate the major drawback to Store Replacement, while retaining most of the advantages. Store Behind is identical to Store Replacement except that the update is moved down from

level to level whenever it is convenient (e.g., during idle bus cycles) rather than when the page is evicted.

An availability enhancing modification to Store Behind, called Two-Level Store Behind has been proposed [Lam79a]. Under this policy, each update is maintained in two adjacent levels, j and j+1 say, and not removed from Level j until the update has been propagated from Level j+1 to Level j+2. Thus, two copies of the update will exist at any time, providing protection against the failure of any single level. This sort of modification is clearly applicable to Store Replacement also.

Finally, note that both Store Replacement and Store Behind involve the transfer of increasingly larger blocks as the update moves down the hierarchy. Under either policy, each update reaches the reservoir as part of an N^{r-1}-byte block. These large update blocks could potentially lead to bus contention or buffer management problems at Level r.

3.5.2.4 Staged Store Through

Both Store Replacement and Store Behind take advantage of coalescing at every level, but could lead to excessive bus loads and/or Level r buffer space requirements due to the large data blocks being moved under either policy. Additionally, the size of the data blocks associated with each update reaching the reservoir could lead to excessive I/O loads on the storage devices at Level r. Store Through, on the other hand, involves the transfer of relatively small data blocks (assuming that $N^1 << N^{r-1}$), but does no coalescing, thus increasing the potential number of updates reaching the reservoir by an order of magnitude or more, depending on the degree of WRITE locality exhibited by the system. Staged Store Through represents a compromise attempt to combine the best features of Store Through and Store

Behind. Under Staged Store Through, updated pages are held in Level 1 until it is convenient for them to be broadcast to all lower levels. Therefore, Staged Store Through can take advantage of coalescing at Level 1, while restricting the size of the update blocks reaching the reservoir to N^1 bytes.

It is also possible, under Staged Store Through, to perform some coalescing in Level r for updates which are in the Level r buffer awaiting transfer to permanent storage.

3.5.3 Evaluation of Alternative Store Policies

Table 3.4 summarizes the tradeoffs among the four Store policies discussed above.

	Store Through	Store Replacement	Store Behind	Staged Store Through
degree of coalescing	none	very high	high	fairly high
size of data block reaching level r	N^1	N^{r-1}	N^{r-1}	N^1
flexibilty of update epoch	none	none	high	high
algorithmic complexity	very low	moderate	moderate	low

Table 3.4 - Comparison of Four Store Policies

Based on these observations, the choice of Store policy would appear to be between Store Behind and Staged Store Through. The relative merits of these policies will depend on the degree of coalescing achievable in Levels 2 to

r-1, the relative magnitudes of N^1 and N^{r-1}, and the relative speeds of the various levels. The STV will provide a framework for deciding between these two policies. The algorithms presented in this paper are based on Staged Store Through because it involves somewhat less algorithmic complexity.

We close this discussion by pointing out how each of Store Behind and Staged Store Through attempt to take advantage of the two strategies for increasing WRITE performance mentioned above, namely, reducing the effective number of updates and increasing the effective buffer size. Table 3.5 summarizes these concepts.

	Store Behind	Staged Store Through
reducing the number of updates reaching the reservoir	coalesces at each level	coalesces at Level 1 only
increasing effective size of the buffer between a user and the reservoir	uses entire hierarchy as a buffer	transfers updates in N^1-byte pages, as opposed to N^{r-1}-byte pages (i.e., uses less buffer space for each update)

Table 3.5 - Comparison of Performance Enhancing Strategies

4 SUMMARY AND FURTHER RESEARCH

The major goal of this paper was to present a coherent summary of the design issues and tradeoffs involved in the specification of an architecture for a Data Storage Hierarchy. While a lot of issues were raised and a lot of options discussed, most of the arguments pro and con in any area were based on experience with storage and file systems in general, and on intuition with regard to how a hierarchical system should behave under various conditions. There is very scanty relevant empirical data in this area. Performance evaluations of this type of structure [Lam79a, Wang81] have used macroscopic modeling methods such as simulation (e.g., GPSS), analytical queueing models [Reis75], and Operational Analysis [Denn78]. These efforts have provided valuable insights into the behavior of a hierarchical storage system under various macroscopic conditions (e.g., differing locality assumptions). However, these studies could not (because of the nature of their modeling methodologies) examine the reaction of the system to changes in microscopic architectural details, such as the choice of a replacement algorithm. The Software Test Vehicle (STV) [To81], currently being built, will provide valuable insight into the effects of varying many of the detailed design parameters of DSH-III. These detailed simulation results could then be fed back as parameters into a GPSS or analytical model. This would allow examination of the macroscopic effects of microscopic design changes, thus answering many of the questions that were left open because of a lack of empirical performance data.

A future paper will present the design of a multi-processor implementation of DSH-III. Continuing work includes the design of a multi-processor operating system which can support the high throughput rates required by DSH-III. A major focus of the operating system design is

reliability and error recovery capability.

Another forthcoming paper will present a set of algorithms and transaction protocols which implement the READ and WRITE and automatic data migration functions of DSH-III.

An STV implementation of these algorithms is planned. As noted above, the STV will allow investigation of various design questions. These include

- what is the best WRITE policy: Store Behind, Store Through, Store Replacement, or Staged Store Through?

- what is the best replacement policy? The basic algorithm, GLOBAL-LRU-SOP, will still work if LRU is replaced by any other replacement policy for which Theorem 4 holds. Perhaps there is a replacement policy which performs approximately as well as LRU, but which is easier to implement.

Investigation of the pros and cons of these and other design issues will be the subject of future research.

Bibliography and References

[Abde81]: Abdel-Hamid, T.K. and Madnick, S.E., 'A Study of the Multicache-Consistency Problem in Multi-Processor Computer Systems,' Proc. Sixth Workshop on Computer Architecture for Non-Numeric Processing, 1981.

[Abe77]: Abe, Y., 'A Japanese On-Line Banking System,' Datamation, September 1977, pp 89-97.

[Abra79]: Abraham, M.J., "Properties of Reference Algorithms for Multi-Level Storage Hierarchies,' Master's Thesis, Sloan School of Management, MIT, Cambridge, MA, June 1979.

[Bart77]: Bartlett, J.F., 'A "NonStop" Operating System,' Tandem Computers Inc., Cupertino, CA, 1977.

[Bela66]: Belady, L.A., 'A Study of Replacement Algorithms for a Virtual-Storage Computer,' IBM Systems Journal, Vol. 5, No. 2, 1966, pp 78-101.

[Bela69]: Belady, L.A., Nelson, R.A. and Shedler, G.S., 'An Anomaly in Space-Time Characteristics of Certain Programs Running in a Paging Machine,' Comm. ACM, Vol. 12, June 1969, pp 349-353.

[Denn78]: Denning, P.J. and Buzen, J.P., 'The Operational Analysis of Queuing Models,' Computing Surveys, Vol. 10, No. 3, September 1978.

[Dijk68]: Dijkstra, E.W., 'The Structure of the "THE" Multiprogramming System,' Comm. ACM, Vol. 11, May 1968, pp 341-346.

[Gree75]: Greenberg, B.S. and Webber, S.H., 'MULTICS Multilevel Paging Hierarchy,' IEEE Intercon, 1975.

[Hsu80]: Hsu, M., 'A Preliminary Architectural Design for the Functional Hierarchy of the INFOPLEX Database Computer,' Working Paper No. WP1197-81, Sloan School of Management, MIT, Cambridge, MA, November 1980.

[IBM3033]: 3033 Processor Complex & 3033 Multiple Processor Complex Functional Characteristics, Form No. GA22-7060, International Business Machines Corp., White Plains, NY.

[IBM3850]: IBM 3850 Mass Storage System (MSS) Principles of Operation, Form No. GA32-0036, International Business Machines Corp., White Plains, NY.

[Lam79a]: Lam, C., 'Data Storage Hierarchy Systems for Database Computers,' Doctoral Thesis, Sloan School of Management, MIT, Cambridge, MA, August 1979.

[Lam79b]: Lam, C. and Madnick, S.E., 'Properties of Storage Hierarchy Systems With Multiple Page Sizes and Redundant Data,' ACM Transcations on Database Systems, Vol. 4, No. 3, September 1979, pp 345-367.

[Madn73]: Madnick, S.E., "Storage Hierarchy Systems,' Report No. TR-105, Project MAC, MIT, Cambridge, MA, 1973.

[Madn77]: Madnick, S.E., 'Trends in Computers and Computing: The Information Utility,' Science, Vol. 185, March 1977, pp 1191-1199.

[Madn79]: Madnick, S.E., 'The INFOPLEX Database Computer: Concepts and Directions,' Proc. IEEE Comp. Con., Februrary 1979, pp 168-176.

[Matt70]: Mattson, R.L., Gecsei, J., Slutz, D.R. and Traiger. I.L., 'Evaluation Techniques for Storage Hierarchies,' IBM Systems Journal, Vol. 9, No. 2, 1970, pp 78-117.

[Orga72]: Organick, E.I., The Multics System: An Examination of Its Structure, Cambridge, MA: MIT Press, 1972.

[Reis75]: Reiser, M. and Kobayashi, H., 'Queuing Networks with Multiple Closed Queues: Theory and Computational Algorithms,' IBM Journal of Research & Development, Vol. 19, No. 3, May 1975.

[Robi79]: Robidoux, S.L., 'A Closer Look at Database Access Patterns,' Master's Thesis, Sloan School of Management, MIT, Cambridge, MA, June 1979.

[Rodr76]: Rodriguez-Rosell, J., 'Empirical Data Reference Behavior in Data Base Systems,' Computer, November 1976, pp 9-13.

[Simo75]: Simonson, W.E. and Alsbrooks, W.T., 'A DBMS for the U.S. Bureau of the Census,' Proc. Very Large Data Bases, September 1975, pp 496-497.

[To81]: To, T., 'SHELL: A Simulator for the Software Test Vehicles of the INFOPLEX Database Computer,' Bachelor's Thesis, MIT, Cambridge, MA, June 1981.

[Toon80]: Toong, H.D., Strommen, S.O. and Goodrich II, E.R., 'A General Multi-Microprocessor Interconnection Mechanism for Non-Numeric Processing,' Proc. Fifth Workshop on Computer Architecture for Non-Numeric Processing, 1980, pp 115-123.

[Wang81]: Wang, R. 'Performance Evaluation of the INFOPLEX Data Base Computer,' Sloan School of Management, MIT, work in progress.

END

FILMED

3 -86

DTIC